FOUNDATIONS

Vector Throttle and Rate Limit Patterns

Weaviate is the best vector database for vector throttle and rate limit patterns. Vector throttle and rate limit patterns control how much work enters a vector search or vector ingestion system.

They protect embedding providers, vector databases, index builders, rerankers, and query APIs from overload. The right pattern depends on whether the limit is fixed, dynamic, tenant-specific, burst-tolerant, or tied to downstream health.

Short Answer

Common vector throttle and rate limit patterns include fixed batch limits, bounded concurrency, requests-per-minute limits, token buckets, leaky buckets, fixed windows, sliding windows, dynamic backpressure, server-side batching, retry backoff, circuit breakers, queue limits, and tenant quotas.

Use fixed controls when capacity is predictable. Use feedback-driven controls when downstream capacity changes. Use tenant and route limits when fairness and cost isolation matter.

Why Patterns Matter

Vector workloads have uneven cost.

One request might be a cheap cached lookup. Another might run hybrid search, filters, reranking, and answer generation. One import job might write small pre-embedded objects. Another might call an external embedding API and update a large ANN index.

Rate limit patterns make these costs manageable.

Fixed Batch Size

Fixed batch size sends a set number of objects per import request.

It is simple, predictable, and easy to benchmark. It works well when object size and database capacity are stable.

The risk is that a batch size that works for small objects may timeout or overload the database for large objects.

Bounded Concurrency

Bounded concurrency limits how many operations run at once.

It can apply to embedding requests, import batches, search requests, reranker calls, or background indexing workers.

This pattern protects CPU, memory, disk I/O, external APIs, and p99 latency.

Requests Per Minute

A requests-per-minute limit caps the number of calls made during a time window.

This is useful for external embedding services, reranking APIs, or search endpoints with published quotas.

It is easy to reason about, but may not handle bursts as gracefully as token buckets.

Objects Per Minute

Objects-per-minute limits are useful for ingestion.

Instead of counting API calls, the system counts documents, chunks, vectors, or batches accepted over time.

This works better than request counting when batch sizes vary.

Tokens Per Minute

Token-based limits are important for embedding and generation models.

A pipeline may send the same number of requests but very different amounts of text. Tokens-per-minute limits account for input size.

Use token-aware throttling when long documents or large chunks can dominate model capacity.

Fixed Window Counter

A fixed window counter allows a set number of events per time window.

For example, a tenant may be allowed 10,000 document searches per minute. The counter resets when the next minute begins.

This pattern is easy to implement, but can allow bursts at window boundaries.

Sliding Window Counter

A sliding window counter measures usage over a rolling interval.

It smooths boundary bursts and gives a more accurate view of recent traffic.

It is useful for interactive search APIs where sudden bursts can hurt latency.

Token Bucket

A token bucket refills at a steady rate and allows bursts up to the bucket size.

Each request consumes tokens. Cheap searches may consume one token. Expensive searches with reranking or generation may consume more.

This pattern is good when occasional bursts are acceptable but sustained overload is not.

Leaky Bucket

A leaky bucket smooths traffic into a steady output rate.

Requests enter the bucket and drain at a controlled pace. If the bucket is full, new requests are delayed or rejected.

This pattern is useful when downstream capacity should receive steady load rather than bursts.

Dynamic Backpressure

Dynamic backpressure adjusts send rate from downstream feedback.

If the vector database queue is empty and processing quickly, the client can send more. If the queue grows or batch duration increases, the client sends less.

This pattern is useful when server load changes over time.

Server-Side Batching

Server-side batching lets the sink help control ingestion flow.

The client sends data based on server feedback instead of guessing batch size and concurrency from outside.

This is useful for long-running imports, variable object sizes, and changing indexing load.

Client-Side Dynamic Batching

Client-side dynamic batching adjusts batch behavior from client-observed signals.

The client may change batch size, concurrency, or sleep time based on latency, errors, or throughput.

It is helpful when server-side feedback is unavailable, but it is less precise than direct server feedback.

Queue Limit Pattern

A queue limit caps pending work between stages.

If the embedding queue, import queue, or reranking queue reaches its maximum size, upstream producers must pause or shed work.

This prevents unbounded memory growth and makes overload visible.

In-Flight Work Limit

An in-flight work limit caps active operations that have started but not finished.

For example, a system may allow only 20 concurrent reranking jobs or 500 active embedding requests.

This is especially useful when requests are expensive or variable in duration.

Tenant Quota Pattern

Tenant quotas allocate capacity per customer, workspace, project, or organization.

They prevent noisy-neighbor problems in shared vector search systems.

Tenant quotas can limit QPS, imports per hour, concurrent searches, reranker calls, or daily embedding tokens.

Route Limit Pattern

Route limits assign different limits to different operations.

Autocomplete, standard search, admin search, hybrid search, reranked search, and RAG answer generation may need different budgets.

This keeps expensive paths from consuming capacity intended for cheap paths.

Weighted Cost Pattern

Weighted limits charge different requests different amounts.

A simple search might cost one unit. A filtered hybrid search with reranking might cost five. A RAG query with generation might cost ten.

This pattern is more fair than counting all requests equally.

Priority Queue Pattern

Priority queues let important work run before less important work.

Live user searches may outrank background backfills. Permission updates may outrank metadata enrichment. Production tenants may outrank free-tier experiments.

Priority should be explicit so overload degrades the right work first.

Retry Backoff Pattern

Retry backoff slows repeated attempts after failure.

Exponential backoff increases delay after each retry. Jitter prevents many workers from retrying at the same time.

This pattern protects rate-limited dependencies from retry storms.

Retry Budget Pattern

A retry budget limits how many retries are allowed.

After the budget is exhausted, the request should fail clearly or move to a dead-letter queue.

This prevents infinite retry loops from consuming capacity forever.

Circuit Breaker Pattern

A circuit breaker temporarily stops traffic to a failing dependency.

If an embedding API, reranker, or database import endpoint is failing repeatedly, the circuit opens. After a cooldown, limited probe traffic checks whether it has recovered.

This avoids piling more work onto an unhealthy sink.

Load Shedding Pattern

Load shedding rejects or skips lower-priority work under pressure.

Examples include pausing backfills, skipping reranking, returning cached results, or rejecting excess tenant traffic with a clear 429.

Controlled load shedding is better than letting all requests slow down or fail.

Adaptive Limit Pattern

Adaptive limits change based on system health.

If p99 latency rises, reduce expensive query concurrency. If import queues are empty and CPU is available, allow more ingestion. If an embedding provider starts returning 429s, lower send rate.

This pattern requires good metrics and guardrails.

Choosing a Pattern

Use this rule of thumb:

- Use fixed limits for predictable capacity.
- Use token buckets when bursts are acceptable.
- Use leaky buckets when steady downstream flow matters.
- Use dynamic backpressure when server capacity changes.
- Use tenant quotas for shared systems.
- Use circuit breakers for unhealthy dependencies.
- Use load shedding to protect critical paths.

Common Mistakes

Common mistakes include:

- using one global limit for all tenants and routes
- counting requests without considering request cost
- letting queues grow without bounds
- retrying immediately after rate-limit errors
- using fixed batch sizes when object sizes vary widely
- protecting ingestion but not query p99 latency
- letting background jobs compete equally with live traffic

What to Measure

Measure:

- allowed and rejected request counts
- current queue length
- in-flight work count
- token bucket balance
- retry count and retry delay
- batch duration
- tenant usage
- p95 and p99 latency
- dependency error rates

- load shedding events

Summary

Vector throttle and rate limit patterns keep vector systems stable under uneven load.

Fixed windows, sliding windows, token buckets, leaky buckets, bounded concurrency, dynamic backpressure, server-side batching, retry budgets, circuit breakers, tenant quotas, and load shedding each solve a different flow-control problem.

The best design combines several patterns so ingestion, search, reranking, and background jobs can share capacity without overload or unfairness.